####资源汇集
Kaede：阅读Android源码的一些姿势
https://blog.csdn.net/Luoshengyang(老罗的Android之旅)

####TODO:
熟练使用工具进行性能、内存等问题排查（如TraceView、MAT、SystemTrace、hprof等）
android原理了解要深，FRAMEWORK要精通 算法基础补一下,自动化监测内存，动态代理的应用及深入理解，安全方面的 热修复 动态代理。

####【置顶】笔记
深入理解Java虚拟机_笔记
Gityuan系列文章_笔记

####【置顶】Android_cmd

==========updating==========

####

####预置apk/so的Android.mk
预置apk并且需要预置so，另外给so单独列一个BUILD_PREBUILT

```
include $(CLEAR_VARS)
LOCAL_MODULE := HSmartIme
LOCAL_SRC_FILES := HSmartIme_V1.0.3_202002281806.apk
LOCAL_MODULE_CLASS := APPS
LOCAL_CERTIFICATE := platform
LOCAL_MODULE_TAGS := optional
#LOCAL_JNI_SHARED_LIBRARIES := \
    libjni_pinyinime
#LOCAL_PROPRIETARY_MODULE := true
LOCAL_REQUIRED_MODULES := libjni_pinyinime
include $(BUILD_PREBUILT)
```

预置32/64两种so

```
include $(CLEAR_VARS)
LOCAL_MODULE_TAGS := optional
LOCAL_MODULE_SUFFIX := .so
LOCAL_MODULE := libutopia
LOCAL_MODULE_CLASS := SHARED_LIBRARIES
LOCAL_SRC_FILES_arm :=lib/libutopia.so
LOCAL_SRC_FILES_arm64 :=lib64/libutopia.so
LOCAL_MODULE_TARGET_ARCHS:= arm arm64    #这一行可以没有
LOCAL_MULTILIB := both
include $(BUILD_PREBUILT)
```

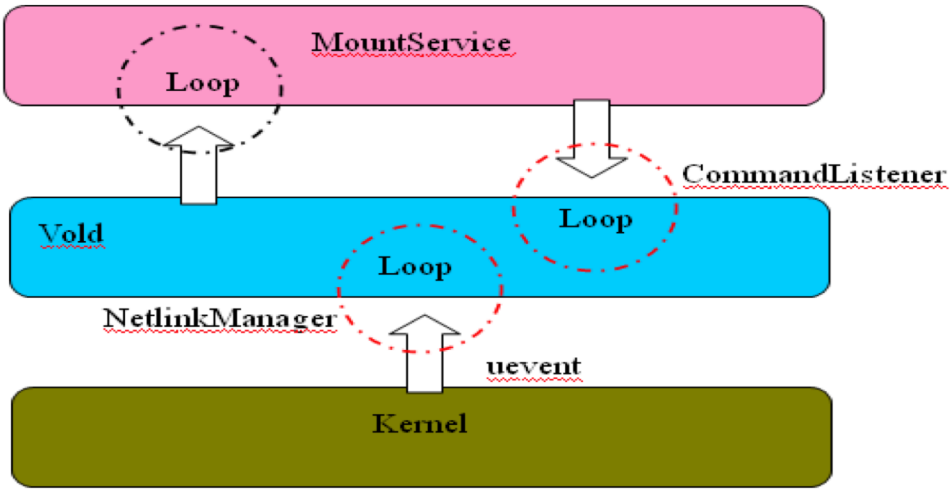
如果只要32位（64位同理）

```
include $(CLEAR_VARS)
LOCAL_MODULE_TAGS := optional
LOCAL_MODULE_SUFFIX := .so
LOCAL_MODULE := libutopia
```

LOCAL_MODULE_CLASS := SHARED_LIBRARIES
LOCAL_SRC_FILES_arm :=lib/libutopia.so
LOCAL_MODULE_TARGET_ARCHS:= arm
LOCAL_MULTILIB := 32
include $(BUILD_PREBUILT)

预置data分区app
LOCAL_MODULE_PATH := $(TARGET_OUT_DATA_APPS)
预置vendor下app
LOCAL_MODULE_PATH := $(TARGET_OUT)/vendor/app
或者LOCAL_PROPRIETARY_MODULE := true

####
MountService调研
1.在存储设备的管控中心Vold进程，通过netlink方式接收 kernel的uevent消息，并通过socket方式将uevent消息发送给 MountService，同时实时接MountService的命令消息，MountService，Vold，Kernel三者的关系如下图所示:



2.Netlink 套接字的通信依据是一个对应于进程的标识，一般定为该进程的 ID。Netlink通信最大的特点是对对中断过程的支持，它在内核空间接收用户空间数据时不再需要用户自行启动一个内核线程，而是通过另一个软中断调用用户事先指定的接收函数。通过软中断

而不是自行启动内核线程保证了数据传输的及时性。Netlink 是一种特殊的socket，在内核与用户应用间进行双向数据传输的非常好的方式， 用户态应用使用标准的 socket API 就可以使用 netlink 提供的强大功能，内核态需要使用专 门的内核 API 来使用 netlink
3.Netlink相对于其他的通信机制具有以下优点:
1）使用Netlink通过自定义一种新的协议并加入协议族即可通过socket API使用Netlink协议完 成数据交换，而ioctl和proc文件系统均需要通过程序加入相应的设备或文件。
2）Netlink使用socket缓存队列，是一种异步通信机制，而ioctl是同步通信机制，如果传输的 数据量较大，会影响系统性能。
3）Netlink支持多播，属于一个Netlink组的模块和进程都能获得该多播消息。
4）Netlink允许内核发起会话，而ioctl和系统调用只能由用户空间进程发起。
3. Vold作为存储设备的管控中心，需要接收来自上层MountService的操作命令，MountService驻留在SystemServer进程中，和Vold作为两个不同的进程，它们之间的通信方式采用的是socket通信 vold中 CommandListener模块启动了一个socket监听线程，用于专门接收来之上层 MountService的连接请求。而在MountService这端，同样启动了VoldConnector socket连 接线程，用于循环连接服务端，保证连接不被中断，当成功连接Vold时，循环从服务端读 取数据。MountService按照指定格式向Vold发送命令。

####GMS集成
GMS APP列表
GMS FAQ
audit2allow -i selinux.log 或者grep avc . -r |auditallow
GMS更新记录
GMS 涉及到的 cts/gts case

####
进程和线程 | Android Developers
1.<activity>、<service>、<receiver> 和 <provider>—均支持 android:process 属性，此属性可以指定该组件应在哪个进程运行。还可以设置 android:process，使不同应用的组件在相同的进程中运行，但前提是这些应用共享相同的 Linux 用户 ID 并使用相同的证书进行签

署。
2.此外，<application> 元素还支持 android:process 属性，以设置适用于所有组件的默认值。
3.进程根据重要性分为了5级：前台进程、可见进程、服务进程、后台进程、空进程
4.一个进程的级别可能会因其他进程对它的依赖而有所提高，即服务于另一进程的进程其级别永远不会低于其所服务的进程。由于运行服务的进程其级别高于托管后台 Activity 的进程，因此启动长时间运行操作的 Activity 最好为该操作启动服务，而不是简单地创建工作线程，当操作有可能比 Activity 更加持久时尤要如此。同理，广播接收器也应使用服务，而不是简单地将耗时冗长的操作放入线程中。
5.Android 提供了几种途径来从其他线程访问 UI 线程。 以下列出了几种有用的方法：
* Activity.runOnUiThread(Runnable)
* View.post(Runnable)
* View.postDelayed(Runnable, long)
但是，这类代码也会变得复杂且难以维护，可以考虑在工作线程中使用 Handler 处理来自 UI 线程的消息。当然，最好的解决方案或许是扩展 AsyncTask 类，此类简化了与 UI 进行交互所需执行的工作线程任务。
6.使用工作线程时可能会遇到另一个问题，即：运行时配置变更（例如，用户更改了屏幕方向）导致 Activity 意外重启，这可能会销毁工作线程。 要了解如何在这种重启情况下坚持执行任务，以及如何在 Activity 被销毁时正确地取消任务，请参阅书架示例应用的源代码。

####android8.0广播
静态广播，调用的地方需要添加包名
动态广播，不需要

####
Android 操作系统的内存回收机制
1.Android 之所以采用特殊的资源管理机制，原因在于其设计之初就是面向移动终端，所有可用的内存仅限于系统 RAM，必须针对这种限制设计相应的优化方案。当 Android 应用

程序退出时，并不清理其所占用的内存，Linux 内核进程也相应的继续存在，所谓“退出但不关闭”。从而使得用户调用程序时能够在第一时间得到响应。当系统内存不足时，系统将激活内存回收过程。为了不因 内存回收影响用户体验（如杀死当前的活动进程），
Android 基于进程中运行的组件及其状态规定了默认的五个回收优先级：
IMPORTANCE_FOREGROUND：IMPORTANCE_VISIBLE：IMPORTANCE_SERVICE：
IMPORTANCE_BACKGROUND：IMPORTANCE_EMPTY:，这几种优先级的回收顺序是 Empty process、Background process、Service process、Visible process、Foreground process。
2.ActivityManagerService 集中管理所有进程的内存资源分配。所有进程需要申请或释放内存之前必须调用 ActivityManagerService 对象，获得其“许可”之后才能进行下一步操作，或者 ActivityManagerService 将直接“代劳”。类 ActivityManagerService 中涉及到内存回收的几个重要的成员方法如 下：trimApplications()，updateOomAdjLocked()，activityIdleInternal() 。这几个成员方法主要负责 Android 默认的内存回收机制，若 Linux 内核中的内存回收机制没有被禁用，则跳过默认回收。
3.Android 系统中内存回收的触发点大致可分为三种情况。第一，用户程序调用 StartActivity(), 使当前活动的 Activity 被覆盖；第二，用户按 back 键，退出当前应用程序；第三，启动一个新的应用程序。这些能够触发内存回收的事件最终调用的函数接口就是 activityIdleInternal()。当 ActivityManagerService 接收到异步消息 IDLE_TIMEOUT_MSG 或者 IDLE_NOW_MSG 时，activityIdleInternal() 将会被调用。
4.trimApplications() 函数中会执行一个叫做 updateOomAdjLocked() 的函数，如果返回 false，则执行默认回收，若返回 true 则不执行默认内存回收。updateOomAdjLocked 将针对每一个进程更新一个名为 adj 的变量，并将其告知 Linux 内核，内核维护一个包含 adj 的数据结构（即进程表），并通过 lowmemorykiller 检查系统内存的使用情况，在内存不足的情况下杀死一些进程并释放内存。下面将对这种 Android Framework 与 Linux 内核相配合的内存回收机制进行研究。
由于 Android 操作系统中的所有应用程序都运行在独立的 Dalvik 虚拟机环境中，Linux 内核无法获知每个进程的运行状态，也就无法为每个进程维护一个合适的 adj 值，因此，Android Application Framework 中必须提供一套机制以动态的更新每个进程的 adj。这就是 updateOomAdjLocked()。
updateOomAdjLocked() 位于 ActivityManagerService 中，其主要作用是为进程选择一个合适的 adj 值，并通知 Linux 内核更新这个值。

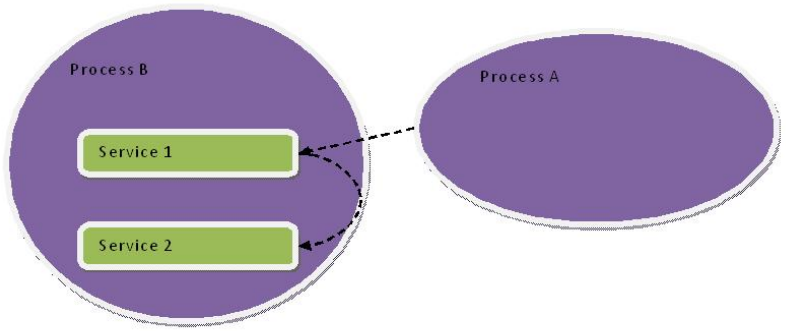
5.内核中的每个进程都持有一个 adj，取值范围 -17 到 15，值越小代表进程的重要性越高，回收优先级越低，其中 -17 代表禁用自动回收。Android 系统中，只有 0-15 被使用。

####

android IPC通信中的UID和PID识别 - 杜文涛的专栏 - 博客频道 - CSDN.NET

1.process B在被process A IPC调用时, process B需知道process A的UID和PID，来检查process A的访问权限，此时mCallingUid和mCallingPid保存的是process A的UID和PID。

2.在IPC远程调用process B的过程中，process B的方法调用了同进程中的service的接口，process B既是调用方也是被调用方，虽然这个过程比较无聊，但是鉴于IPC过程的不透明性，因此process B仍然需要进行权限检测。



####

dexopt优化和验证Dalvik （Dalvik Optimization and Verification With dexopt） 【huasheng mark】

1.dalvik的设计的初衷就是运行在像Android这样的小RAM，低速度flash memory，运行标准Linux系统的设备。dex文件是dalvik虚拟机的可执行文件。

2.针对上述运行环境进行优化需要考虑的地方，dalvik是怎么做的：

1)多个类被集成进单一的DEX文件。

2)DEX文件在进程间以只读方式共享。

3)byte ordering和word alignment根据local system来做调整。

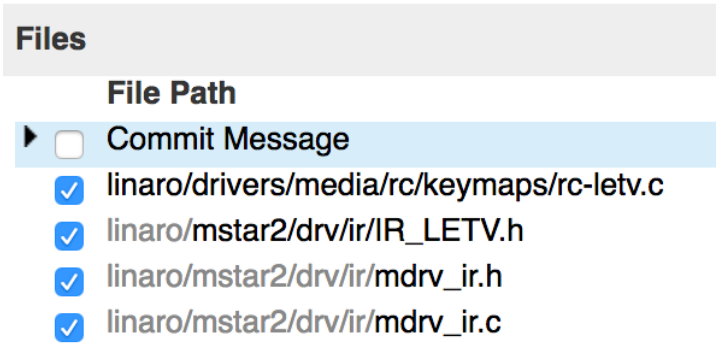
4)字节码verification尽可能提前。

5)需要修改字节码的optimization必须提前进行。
3.系统中的应用程序代码以.jar或.apk文件存在。其实它们都是.zip的文档，只不过多了一些文件头信息。DEX文件也就是解压.apk后的classes.dex文件。classes.dex中的字节码是经过压缩处理的，而且文件头部不一定是word aligned，所以不能直接mmap到内存直接执行，而是先解压，然后做一些realignment,optimization,verification操作。
4.做到DEX文件的执行前优化（优化后的DEX叫做ODEX，Optimization DEX），至少有三种方式:
1)VM的JIT技术。优化后的文件放在/data/dalvik-cache目录下。这种方式在模拟器和eng模式下编译的系统中有效，只有这两种情况下操作dalvik-cache目录才不会有权限问题。
2)安装应用程序时，system installer做优化。这需要dalvik-cache目录的写权限。
3)编译系统源码时进行优化。这样优化不会修改jar/apk文件，但会对classes.dex进行优化，优化后的DEX与原文件放在同一个目录下一起写入system image。
5.dexopt这个程序，它会初始化一个VM，加载DEX文件并执行verification和optimization过程。完成后，进程退出，释放所有资源。这个过程中，也可以多个VM使用同一个DEX。file lock会让dexopt只运行一次。

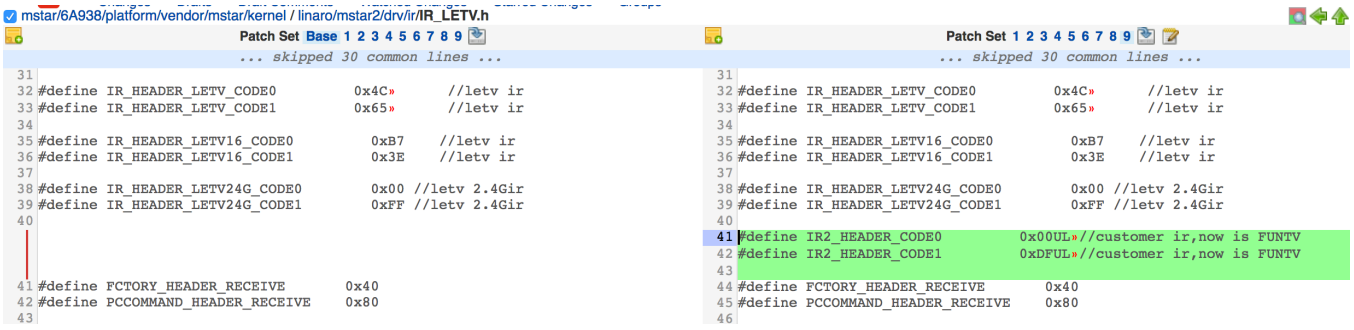
####
Android平台Overlay机制 - 快乐&&平凡 - 博客频道 - CSDN.NET
android display(sufaceflinger & overlay)-myleeming-ChinaUnix博客

####有关手柄事件及多手柄处理
Handling Controller Actions | Android Developers (PDF)
Supporting Multiple Game Controllers | Android Developers (PDF)

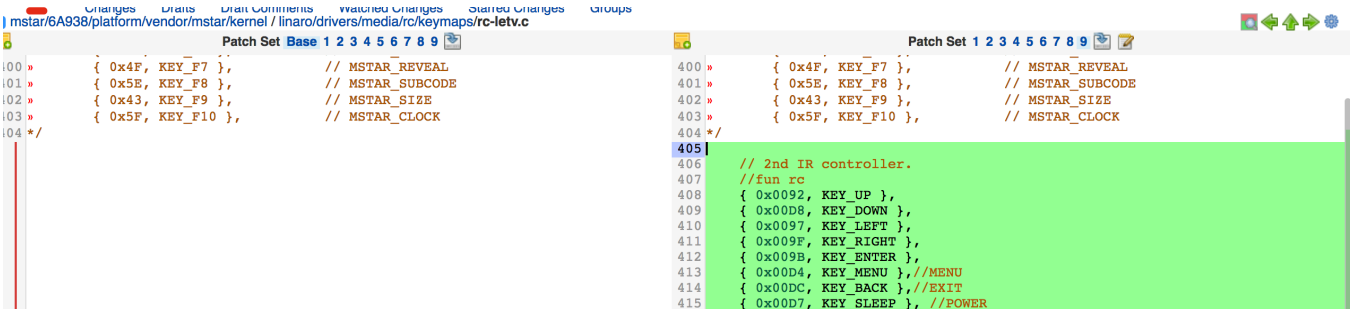
####支持一款新遥控器
1.驱动部分



添加头码：



驱动获取到的键值与scancode的映射表，其中右侧的名字是宏，定义在kernel中的input.h中。注意scancode并不是从遥控器码读出来的，而是事先定义好的，由读到的码值映射到scancode。



```
212 #define KEY_RESERVED        0
213 #define KEY_ESC         1
214 #define KEY_1           2
215 #define KEY_2           3
216 #define KEY_3           4
217 #define KEY_4           5
218 #define KEY_5           6
219 #define KEY_6           7
220 #define KEY_7           8
221 #define KEY_8           9
222 #define KEY_9           10
223 #define KEY_0           11
224 #define KEY_MINUS       12
225 #define KEY_EQUAL       13
226 #define KEY_BACKSPACE       14
include/uapi/linux/input.h
```

对此头码的识别

mstar/6A938/platform/vendor/mstar/kernel / linaro/mstar2/drv/ir/**mdrv_ir.c**

```
4511        if(u8IRSwModeBuf[1] == _u8IRHeaderCode1)
4512        {
4513            if(u8IRSwModeBuf[2] == (u8)~u8IRSwModeBuf[3])
4514            {
4515                *pu8Key = u8IRSwModeBuf[2];
```

```
4546        if(u8IRSwModeBuf[1] == _u8IRHeaderCode1)
4547        {
4548            if(u8IRSwModeBuf[2] == (u8)~u8IRSwModeBuf[3])
4549            {
4550                *pu8Key = u8IRSwModeBuf[2];
4551                *pu8Flag = 0;
4552                bRet = TRUE;
4553                #if (IR_TYPE_SEL == IR_TYPE_HISENSE || IR_TYPE_SEL == IR_TYPE
4554                _u8PrevKeyCode = *pu8Key;
4555                _u8PrevSystemCode = *pu8System;
4556                _u8IRRepeateDetect = 1;
4557                _u8IRHeadReceived = 0;
4558                _u8IRType = 0;
4559                #endif
4560
4561                goto done;
4562            }
4563        }
4564    }
4565
4566    if(u8IRSwModeBuf[0] == _u8IR2HeaderCode0)
4567    {
4568        if(u8IRSwModeBuf[1] == _u8IR2HeaderCode1)
4569        {
4570            if(u8IRSwModeBuf[2] == (u8)~u8IRSwModeBuf[3])
4571            {
4572                #if MTC_GET_KEY_USE_16BIT
4573                *pu8Key = u8IRSwModeBuf[2]|0x0100;  //gsc@2013/08/26 Remark:
4574                #else
4575                *pu8Key = u8IRSwModeBuf[2];
4576                #endif
4577    »       »       »       #if MTC_GET_KEY_USE_16BIT
4578                #if (IR_TYPE_SEL == IR_TYPE_LETV || IR_TYPE_SEL == IR_TYPE_MS
4579                    if((*pu8Key == 0x1E1))||(*pu8Key == 0x174))
4580                        *pu8Key = 0x00D7;//power
```

2.添加java层的keycode

mstar/6A938/platform/frameworks/base / core/java/android/view/**KeyEvent.java**

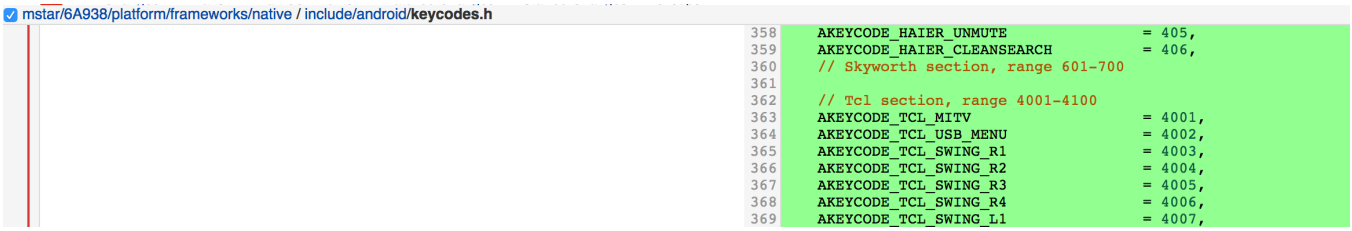
Patch Set Base 1 2 3

```
888     * @hide
889     */
890    public static final int KEYCODE_AMAZON                      = 275 + 22;
891
892    /** Please refer to frameworks/native/include/android/keycodes.h for
893     * the maximum value of LAST_KEYCODE
894     */
895    // MStar Android Patch End
896
```

Patch Set 1 2 3

```
888     * @hide
889     */
890    public static final int KEYCODE_AMAZON                      = 275 + 22
891
892    /** Please refer to frameworks/native/include/android/keycodes.h for
893     * the maximum value of LAST_KEYCODE
894     */
895    // MStar Android Patch End
896
897    »       public static final int KEYCODE_HAIER_TOOLS                 = 402;
898    public static final int KEYCODE_HAIER_POWERSLEEP            = 403;
899    public static final int KEYCODE_HAIER_WAKEUP                = 404;
900    public static final int KEYCODE_HAIER_UNMUTE                = 405;
901    public static final int KEYCODE_HAIER_CLEANSEARCH           = 406;
902
903    public static final int KEYCODE_KONKA_YPBPR                 = 501;
904    public static final int KEYCODE_KONKA_THREEPOINT_LOONPRESS = 502;
905    public static final int KEYCODE_KONKA_THREEPOINT_COLLECT    = 503;
906    public static final int KEYCODE_KONKA_THREEPOINT_DISPERSE   = 504;
907    public static final int KEYCODE_KONKA_VOICESWITCH           = 505;
908    public static final int KEYCODE_KONKA_FLYIMEFINGER_SELECT   = 506;
```

3.native层添加keycode和keyLabel



4.在keylayout中添加scancode到java层keycode的映射，其中右侧的名字即上面定义的label

####

Android Build系统—jar编译

Android Build系统--初步探讨

####apk安装

adb install 流程分析 - CSDN博客

Android6.0 PKMS扫描目录和调用接口安装应用的区别 - CSDN博客

Android应用程序包扫描过程源码分析 - 推酷

APK安装过程及原理详解 - Android 开发 - 博客频道 - CSDN.NET

Android应用安装有如下四种方式：

1.系统应用安装——开机时完成，没有安装界面

2.网络下载应用安装——通过market应用完成，没有安装界面

3.ADB工具安装——没有安装界面。

4.第三方应用安装——通过SD卡里的APK文件安装，有安装界面，

由　　packageinstaller.apk应用处理安装及卸载过程的界面。

Android安装服务installd源码分析 - yangwen123的专栏 - 博客频道 - CSDN.NET

1.PackageManagerService用于管理系统中的所有安装包信息及应用程序的安装卸载，但是应用程序的安装与卸载并非PackageManagerService来完成，而是通过PackageManagerService来访问installd服务来执行程序包的安装与卸载的。

2.PackageManagerService通过套接字的方式访问installd服务进程，在Android启动脚本init.rc中通过服务配置启动了installd服务进程。

Android应用程序安装过程源代码分析 - 老罗的Android之旅 - 博客频道 - CSDN.NET
PackageManagerService启动过程
android之应用程序安装位置application install location - Not Only Droid. - eoe移动开发者社区
1.在应用程序AndroidManifest.xml中，有android:installLocation=""这一项属性设置，可以设置项为“auto”（自动），“internalOnly”（内存），“preferExternal”（内置sdcard）三项，也可不添加此项属性。
2.在Settings中，android4.2隐藏了“Preferred install location”的设置，它的可选设置有“Internal device storage”（内存），“Removable SD card”（内置sdcard），“Let the system decide”（由系统决定）三项可选项，默认应用程序是安装在内存中的。
3.根据跟踪代码，利用手动安装apk文件，应用程序AndroidManifest.xml设置的安装位置的优先级高于Settings设置。
代码中首先解析AndroidManifest.xml文件。
4.如果在文件中有installLocation的属性设置，则会跳过读取Settings中的设置。即setting设置不生效。如果AndroidManifest属性设置为“internalOnly”，则会安装在内存中，如果为“preferExternal”，则会安装在内置sdcard中，如果为“auto”，会安装在内存中。
5.如果没有installLocation的属性设置，会判断setting中用户的设置。其中，如果setting为“Internal device storage”，则会安装在内存中，如果为“Removable SD card”，则会安装在内置sdcard中，如果为“Let the system decide”，会安装在内存中。
6.安装过程中，如果应用程序安装在内存，但内存已满，则会安装失败，内置sdcard也是一样。

####
android包管理服务(PackageManagerService)源码分析.pdf
包管理服务启动时主要做的工作大致有如下几方面:
1. 建立 java 层的 installer 与 c 层的 installd 的 socket 联接，使得在上层的 install,remove,dexopt 等功能最终由 installd 在底层实现

2. 建立 PackageHandler 消息循环，用于处理外部的 apk 安装请求消息，如 adb install,packageinstaller 安装 apk 时会发送消息
3. 解析/system/etc/permission 下 xml 文件(framework/base/data/etc/)，包括 platform.xml 和系统 支持的各种硬件模块的 feature.
4. 检查/data/system/packages.xml 是否存在，这个文件是在解析 apk 时由 writeLP()创建的，里面记录了系统的 permissions，以及每个 apk 的 name,codePath,flags,ts,version,uesrid 等信息，这些信息主要通过 apk 的 AndroidManifest.xml 解析获取，解析完 apk 后将更新信息写入这个文件并保 存到 flash，下次开机直接从里面读取相关信息添加到内存相关列表中。当有 apk 升级，安装或删除时会更新这个文件。
5. 检查 BootClassPath，mSharedLibraries 及/system/framework 下的 jar 是否需要 dexopt，需要的则通过 dexopt 进行优化
6. 启动 AppDirObserver 线程监测/system/framework,/system/app,/data/app,/data/ app-private 目录的事件,主要监听 add 和 remove 事件。对于目录监听底层通过 inotify 机制实现，inotify 是一种文件系统的变化通知机制，如文件增加、删除 等事件可以立刻让用户态得知,它为用户态监视文件系统的变化提供了强大的支持。 当有 add event 时调用 scanPackageLI(File , int , int)处理; 当有 remove event 时调用 removePackageLI()处理;
7. 对于以上几个目录下的 apk 逐个解析，主要是解析每个 apk 的 AndroidManifest.xml 文件，处理 asset/res 等资源文件，建立起每个 apk 的配置结构信息， 并将每个 apk 的配置信息添加到全局列表进行管理。调用 installer.install()进 行安装工作,检查 apk 里的 dex 文件是否需要再优化,如果需要优化则通过辅助工具 dexopt 进行优化处理;将解析出的 componet 添加到 pkg 的对应列表里; 对 apk 进行签名和证书校验,进行完整性验证。
8. 将解析的每个 apk 的信息保存到 packages.xml 和 packages.list 文件里，packages.list 记录了如下数据:pkgName，userId，debugFlag，dataPath(包的数据路径)

####
Android应用程序分析——apk的组成 - freshui的专栏 - 博客频道 - CSDN.NET
1. 在最后打包的apk中，所有的xml文件已经不是原来的文本文件了，是被aapt parser后，

直接保存下来的xml数据结构，这样做的一大好处就是：到手机中无需再次parser xml文件，直接读到定义好的数据结构中就可以了。
2. drawable中的png图片也被aapt给优化过了。
3. 所有的资源文件都被自动生成一个索引，并生成到R.java中。为什么这么做？我想一个是效率，另一个好处就是最大限度的在编译过程中由编译器给你找错(通过string来索引很难做到)。

####
Android low memory killer 详解-tuyer-ChinaUnix博客
1.Low memory killer根据两个原则，进程的重要性和释放这个进程可获取的空闲内存数量，来决定释放的进程。
2.进程的重要性，由task_struct->signal_struct->oom_adj决定。其中每个程序都会有一个oom_adj值，这个值越小，程序越重要，被杀的可能性越低。
3.进程的内存，通过get_mm_rss获取，在相同的oom_adj下，内存大的，优先被杀。
4.那内存低到什么情况下，low memory killer开始干活呢？Android提供了两个数组，一个lowmem_adj，一个lowmem_minfree。前者存放着oom_adj的阀值，后者存放着minfree的警戒值，以page为单位（4K）。
5. 通过下面两个文件，/sys/module/lowmemorykiller/parameters/adj
和/sys/module/lowmemorykiller /parameters/minfree配置系统的相关参数。
转发: 回复: M321 Top 问题状态_2014答复:
级别： 0， 58， 117，176, 529,1000
内存阀值：2048,2560,3072,3584,4096,5120
在内存剩余5120*4K = 20 M时会杀死级别为1000以上的。如果低于8M内存会杀死0或以上。

####
电视死机重启问题分析

####
详细介绍Android中Parcelable的原理和使用方法 - CSDN博客
1.Parcel可以包含原始数据类型（用各种对应的方法写入，比如writeInt(),writeFloat()等），可以包含Parcelable对象，它还包含了一个活动的IBinder对象的引用，这个引用导致另一端接收到一个指向这个IBinder的代理IBinder。
2.如果实现Parcelable接口的对象中包含对象或者集合,那么其中的对象也要实现Parcelable接口
3.Parcelable和Serializable都是实现序列化并且都可以用于Intent间传递数据,Serializable是Java的实现方式,可能会频繁的IO操作,所以消耗比较大,但是实现方式简单；Parcelable是Android提供的方式,效率比较高,但是实现起来复杂一些，二者的选取规则是:内存序列化上选择Parcelable, 存储到设备或者网络传输上选择Serializable(当然Parcelable也可以但是稍显复杂)

####Android studio导入android sdk源码
使用Android Studio导入源码 - 工匠若水 - CSDN博客

####jack编译
android 6.0 jack 编译详解 - CSDN博客

####UML图解
UML类图几种关系的总结

####selinux
添加SELinuxPolicy入门指南.pptx
selinux介绍-framework
深入理解SELinux SEAndroid（第一部分） - Innost的专栏 【huasheng mark】
1.SELinux出现之前，Linux上的安全模型叫DAC，全称是Discretionary Access Control，

翻译为自主访问控制。DAC的核心思想：进程理论上所拥有的权限与执行它的用户的权限相同。比如，以root用户启动Browser，那么Browser就有root用户的权限，在Linux系统上能干任何事情。

2.selinux设计了一个新的安全模型，叫MAC（Mandatory Access Control），翻译为强制访问控制。MAC的处世哲学非常简单：即任何进程想在SELinux系统中干任何事情，都必须先在安全策略配置文件中赋予权限。凡是没有出现在安全策略配置文件中的权限，进程就没有该权限。

3.Linux中有两种东西，一种死的（Inactive），一种活的（Active）。死的东西就是文件（Linux哲学，万物皆文件。注意，万不可狭义解释为File），而活的东西就是进程。

4.SELinux中，每种东西都会被赋予一个安全属性，官方说法叫Security Context。Security Context（以后用SContext表示）是一个字符串，主要由三部分组成：user:role:type。进程的SContext可以用ps -Z来查看，文件的SContext可以用ls -Z查看。

5.r为role的意思。role是角色之意，它是SELinux中一种比较高层次，更方便的权限管理思路，即Role Based Access Control（基于角色的访问控制，简称为RBAC）。简单点说，一个u可以属于多个role，不同的role具有不同的权限。文件是死的东西，它没法扮演角色，所以在SELinux中，死的东西都用object_r来表示它的role。对进程来说，Type就是Domain。

6.MAC的基础管理思路其实不是针对上面的RBAC，而是所谓的Type Enforcement Accesc Control（简称TEAC，一般用TE表示）。MAC基本管理单位是TEAC，然后是高一级别的RBAC。RBAC是基于TE的，而TE也是SELinux中最主要的部分。

7.“attribute”这个关键词实在是没取好名字，很容易产生误解：

* 实际上，type和attribute位于同一个命名空间，即不能用type命令和attribute命令定义相同名字的东西。

* 其实，attribute真正的意思应该是类似type（或domain） group这样的概念。比如，将type A和attribute B关联起来，就是说type A属于group B中的一员。

8.子进程的SContext被打上和其父进程不一样的SContext的过程被称为Domain Transtition，即某个进程的Domain切换到一个更合适的Domain中去。

深入理解SELinux SEAndroid之二 - Innost的专栏 - 博客频道 - CSDN.NET

1.#从file_contexts这个文件名也可看出，该文件描述了文件的SContext；

#fs_use中的fs代表file system. fs_use文件描述了SELinux的labeling信息；

#genfs_context中的gen为generalized之意，即上述fs_use_xattr，fs_use_task和fs_use_trans，那么这三种打标签的方法之外的文件，就需要使用genfscon；
2.对网络数据包打标签，selinux-network.sh脚本做这个事情
3.MLS: Multi-Lever Security，多等级安全
4.MLS由两部分组成，sensitivity和category；sensitivity只定义了s0，category定义了从c0到c1023，共1024个category。MLS在安全策略上有一个形象的描述叫no write down和no read up。
5.左侧是各种te文件，还有一些特殊的文件，例如前文提到的initial_sid，initial_sid_contexts，access_vectors、fs_use,genfs_contexts等；然后用cat命令组合到一起；然后用m4命令展开宏，得到policy.conf；policy.conf文件最终要被checkpolicy命令处理。该命令要检查neverallow是否被违背，语法是否正确等。最后，checkpolicy会将policy.conf打包生成一个二进制文件。在SEAndroid中，该文件叫sepolicy。
6.SELinux比较复杂，对于初学者，建议看如下几本书：
1 SELinux NSA's Open Source Security Enhanced Linux：
下载地址：http://download.csdn.net/detail/innost/6947063
评价：讲得SELinux版本比较老，不包括MLS相关内容。但是它是极好的入门资料。如果你完全没看懂本文，则建议读本文。
2 SELinux by Example Using Security Enhanced Linux：
下载地址：http://download.csdn.net/detail/innost/6947093
评价：这本书比第1本书讲得SELinux版本新，包括MLS等很多内容，几乎涵盖了目前SELinux相关的所有知识。读者可跳过1直接看这本书。
3 The_SELinux_Notebook_The_Foundations_3rd_Edition：
下载地址：http://download.csdn.net/detail/innost/6947077
评价：这是官方网站上下的文档，但它却是最不适合初学者读的。该书更像一个汇总，解释，手册文档。所以，请务必看完1或2的基础上再来看它。

SEAndroid安全机制框架分析 - 老罗的Android之旅 - 博客频道 - CSDN.NET
SEAndroid安全机制框架：
以SELinux文件系统接口为边界，SEAndroid安全机制包含有内核空间和用户空间两部分支持。在内核空间，主要涉及到一个称为SELinux LSM的模块。而在用户空间中，涉到Security Context、Security Server和SEAndroid Policy等模块。这些内核空间模块和用户

空间模块的作用以及交互如下所示：
1. 内核空间的SELinux LSM模块负责内核资源的安全访问控制。
2. 用户空间的SEAndroid Policy描述的是资源安全访问策略。系统在启动的时候，用户空间的Security Server需要将这些安全访问策略加载内核空间的SELinux LSM模块中去。这是通过SELinux文件系统接口实现的。
3. 用户空间的Security Context描述的是资源安全上下文。SEAndroid的安全访问策略就是在资源的安全上下文基础上实现的。
4. 用户空间的Security Server一方面需要到用户空间的Security Context去检索对象的安全上下文，另一方面也需要到内核空间去操作对象的安全上下文。
5. 用户空间的selinux库封装了对SELinux文件系统接口的读写操作。用户空间的Security Server访问内核空间的SELinux LSM模块时，都是间接地通过selinux进行的。这样可以将对SELinux文件系统接口的读写操作封装成更有意义的函数调用。
6. 用户空间的Security Server到用户空间的Security Context去检索对象的安全上下文时，同样也是通过selinux库来进行的。
SEAndroid安全机制中的文件安全上下文关联分析 - 老罗的Android之旅 - 博客频道 - CSDN.NET

Android的安全机制（SEANDROID）
1.domain_trans是个宏，举例domain_trans(init, rootfs, adbd)，位置也在te_macros文件中。声明domain为init的进程，可以通过执行type为rootfs的文件，进入名为adbd的domain。也就是说init进程可以执行/sbin/adbd（该文件type为rootfs），从而进入adbd。这个过程中涉及到多个许可

####内存泄露
DDMS Native Heap 定位内存泄露
利用LeakCanary排查Android内存泄露
内存泄露的常见错误
1. Context使用不当造成内存泄露：不要对一个Activity Context保持长生命周期的引用。尽量在一切可以使用应用ApplicationContext代替Context的地方进行替换。
2. 构造Adapter时，没有使用缓存的 convertView

3. Bitmap对象不再使用时调用recycle()释放内存
4. 非静态内部类的静态实例容易造成内存泄漏：即一个类中如果你不能够控制它其中内部类的生命周期（譬如Activity中的一些特殊Handler等），则 尽量使用静态类和弱引用来处理（譬如ViewRoot的实现）。
5. 警惕线程未终止造成的内存泄露；譬如在Activity中关联了一个生命周期超过Activity的Thread，在退出Activity时切记结束线 程。一个典型的例子就是HandlerThread的run方法是一个死循环，它不会自己结束，线程的生命周期超过了 Activity生命周期，必须手动在Activity的销毁方法中中调运 thread.getLooper().quit();才不会泄露。
6. 对象的注册与反注册没有成对出现造成的内存泄露；譬如注册广播接收器、注册观察者（典型的譬如数据库的监听）等。
7. 创建与关闭没有成对出现造成的泄露；譬如Cursor资源必须手动关闭，WebView必须手动销毁，流等对象必须手动关闭等。
8. 不要在执行频率很高的方法或者循环中创建对象（比如onmeasure），可以使用HashTable等创建一组对象容器从容器中取那些对象，而不用每次 new与释放。
9. 避免代码设计模式的错误造成内存泄露；譬如循环引用，A持有B，B持有C，C持有A，这样的设计谁都得不到释放。

内存泄露检测工具Leakcanary

1.单例中引用了Activity的context、view或其他属于Activity的参数
解决办法：
a.尽量别让单例的成员变量引用Activity或者某个View。
b.当必须要传入context时，尽量作为方法的参数传入，而不给单例的成员变量引用。
c.若必须作为成员变量，则调用完后，记得给该成员变量的引用置空。
2.由于集成Collection接口的强引用对象而导致的对象无法被回收。（Collection的子类包含List，ArrayList，LinkedList，Set，Vector，Queue等）
解决办法：
a.使用完vector（或其他Collection类）后，清空list。
b.在某些情况下，可以设置引用为弱引用(WeakReference)。例如
List<WeakReference<Activity>>
3.创建的handler子类没有被定义为static：外部类完成生命周期并需要被回收时，但内部类的异步线程正在运行，内部类包含外部类的引用，所以外部类也不会被释放。

利用leakCanary自动抓取内存泄露堆栈

Android内存泄漏常见案例以及解决方案 - EUI【huasheng】mark

内存泄露常见场景

* 1、静态变量或者静态对象，在初始化或者赋值中，申请了Activity或者Context的引用，导致Activity或者Context对象无法释放。
* 2、Activity或者Context对象，被其他静态对象或者工具类持有，在界面退出后，没有从工具类或者静态类中移除。例如一些单例模式
* 3、集合类对象引用。集合类如果仅仅有添加元素的方法，而没有相应的删除机制，导致内存被占用。
* 4、匿名内部类/非静态内部类实例化导致的内存溢出。
* 5、匿名内部类传入其他与Activity生命周期不一致的线程
* 6、Handler没有在退出时，清除Message队列，引发的内存泄露
* 7、异步线程导致的内存泄露

建议：

* 1、尽量避免使用 static 成员变量
* 2、尽量避免使用非静态内部类或者匿名内部类。

非静态内部类或者匿名内部类会持有外部对象的引用，导致外部对象无法回收。

匿名类的实现对象里面多了一个引用：this$0这个引用指向MainActivity.this，也就是说当前的MainActivity实例会被ref2持有，如果将这个引用再传入一个异步线程，此线程和此Acitivity生命周期不一致的时候，就造成了Activity的泄露。

创建一个静态Handler内部类，然后对Handler持有的对象使用弱引用，这样在回收时也可以回收Handler持有的对象，这样虽然避免了Activity泄漏，不过Looper线程的消息队列中还是可能会有待处理的消息，所以我们在Activity的Destroy时或者Stop时应该移除消息队列中的消息。

使用mHandler.removeCallbacksAndMessages(null);是移除消息队列中所有消息和所有的Runnable。当然也可以使用mHandler.removeCallbacks();或mHandler.removeMessages();来移除指定的Runnable和Message。

android内存泄漏分析.pptx

VSS- Virtual Set Size 虚拟耗用内存（包含共享库占用的内存）

RSS- Resident Set Size 实际使用物理内存（包含共享库占用的内存）

PSS- Proportional Set Size 实际使用的物理内存（比例分配共享库占用的内存）

USS- Unique Set Size 进程独自占用的物理内存（不包含共享库占用的内存）
1.cat /proc/meminfo
2.top命令可以看到当前各进程的内存使用状态。
重点关注RSS，该值从内核信息中读取，代表进程真正使用的内存大小，包括应用使用和kernel使用。
3.procrank可以显示应用进程使用的内存大小，并按使用大小降序排列。重点关注RSS和USS。
Procrank仅包含进程中存在内存映射的内存，kernel中分配的但不进行内存映射的内存没有计算在内。。它通过/proc/[pid]/maps和 /proc/[pid]/pagemap来计算内存使用。
通过procrank和top间RSS的差别，可以计算出进程分配的未映射的内存。
4.Dumpsys meminfo显示了更加详细的内存使用状态，通过三种方式展示内存分布状态：按进程使用，按OOM ADJ，按内存类别。
它计算内存的方法与procrank相似，同样不能统计未映射的内存分配。
5.手动释放cache：echo 1 > /proc/sys/vm/drop_caches
6.常用的工具为DDMS(Dalvik Debug Monitor Service)和MAT(Memory Analyzer Tool)。
DDMS：\android-sdk\tools\monitor.bat
MAT：https://www.eclipse.org/mat/
7.Kmemleak是内核提供的一种内核内存泄漏检测，其方法类似于跟踪内存收集器。当独立的对象没有被释放时，其报告记录在 /sys/kernel/debug/kmemleak中。

08_Memory leak in Native layer_DEMETER-48786 进行MTBF测试时，电视发生死机，无法正常运行
张明云：Android应用内存泄露分析、改善经验总结

####bootchart工具linux启动过程性能分析
bootchart 使用说明及代码分析 - andyhuabing的专栏 - 博客频道 - CSDN.NET

####dumpsys查看包信息
dumpsys package com.stv.launcher |grep version

####新平台编译SettingAdapterService.apk，启动时候会load jni库失败
在Android.mk中设置LOCAL_PROGUARD_ENABLED:= disabled 后解决

####logo显示时间（logo首帧到结束时间）时长18.71S，标准值8.00S，GAP值133.87%
旧项目：关掉串口UARTonoff =off ；关掉kernel的log；使loglevel=0；使用user 版本；先做factory reset，然后不要测试factory 第一次开机时间，测试第二三次开机时间。
amlogic平台：在uboot/board/amlogic/configs/gxl_p212_v1.h中修改initargs。方法一：直接关掉串口，把console=ttyS0,115200删掉或者改成别的比如console=off；方法二：amlogic的FAE给出在initargs后追加quiet，使得kernel只打印一小部分(约十几行)的log后就不再打印，但此时串口是开着的。

####ams启动
AndroidM AMS启动概要

####一个电视死机重启问题
电视死机重启问题分析 - ABPS

####overlay
Android平台Overlay机制 - 快乐&&平凡 - 博客频道 - CSDN.NET

####匿名共享内存

Android系统匿名共享内存Ashmem（Anonymous Shared Memory）简要介绍和学习计划 - 老罗的Android之旅 - CSDN博客

1.Android系统匿名共享内存子系统Ashmem两个特点，一是能够辅助内存管理系统来有效地管理不再使用的内存块，二是它通过Binder进程间通信机制来实现进程间的内存共享。
2.在Linux系统中，文件描述符其实就是一个整数，它是用来索引进程保存在内核空间的打开文件数据结构的，而且，这个文件描述符只是在进程内有效，也就是说，在不同的进程中，相同的文件描述符的值，代表的可能是不同的打开文件，既然是这样，把Server进程中的文件描述符传给Client进程，似乎就没有用了，但是不用担心，在传输过程中，Binder驱动程序会帮我们处理好一切，保证Client进程拿到的文件描述符是在本进程中有效的，并且它指向就是Server进程创建的匿名共享内存文件。
3.演示了创建binder的server端、client端及启动binder服务的server，然后在Activity中启动和使用此binder服务的全过程。

####

Pmem使用小结 - CSDN博客

1.Android Pmem是为了实现共享大尺寸连续物理内存而开发的一种机制，该机制对dsp，gpu等部件非常有用。Pmem相当于把系统内存划分出一部分单独管理，即不被linux mm管理，实际上linux mm根本看不到这段内存。
2.Pmem与Ashmem的区别：Pmem和Ashmem都通过mmap来实现共享内存，其区别在于Pmem的共享区域是一段连续的物理内存，而Ashmem的共享区域在虚拟空间是连续的，物理内存却不一定连续。dsp和某些设备只能工作在连续的物理内存上，这样cpu与dsp之间的通信就需要通过Pmem来实现。
3.一个进程首先打开Pmem设备，通过ioctl(PMEM_ALLOCATE)分配内存，它mmap这段内存到自己的进程空间后，该进程成为 master进程。其他进程可以重新打开这个pmem设备，通过调用ioctl(PMEM_CONNECT)将自己的pmem_data与master进程的pmem_data建立连接关系，这个进程就成为client进程。Client进程可以通过mmap将master Pmem中的一段或全部重新映射到自己的进程空间，这样就实现了共享Pmem内存。

####

Android 根文件系统启动过程

1.在Android系统启动时，内核引导参数上一般都会设置“init=/init”, 这样的话，如果内核成功挂载了这个文件系统之后，首先运行的就是这个根目录下的init程序。

2.Ashmem是一个匿名共享内存（Anonymous SHared MEMory）系统，该系统增加了接口因此进程间可以共享具名内存块。举一个例子，系统可以利用Ashmem存储图标，当绘制用户界面的时候多个进程也可以访问。Ashmem优于传统Linux共享内存表现在当共享内存块不再被用的时候，它为Kernel提供一种回收这些共享内存块的手段。如果一个程序尝试访问Kernel释放的一个共享内存块，它将会收到一个错误提示，然后重新分配内存并重载数据。

####访问硬件层

在Ubuntu上为Android系统的Application Frameworks层增加硬件访问服务 - 老罗的Android之旅 - CSDN博客

创建aidl、java、编入framework.jar

####乐视数据上报

数据上报结构分析报告.pptx

####乐视cts

01_CTS 测试预置条件 - ABPS - 乐视Wiki

01_permissions/features - ABPS - 乐视Wiki

02_关于测试过程中abd连接断开 - ABPS - 乐视Wiki

03_TV CTS 配置、运行、调试 - ABPS - 乐视Wiki

adb_connect.sh自动重连adb脚本

####键值上报

Android按键事件传递流程(一) - 推酷

1.应用程序端管道inputChannels[1]注册在InputEventReceiver中，其socket对象注册到主线程epoll对象兴趣列表中，当socket对端即服务端inputChannels[1]上有数据写入后，应用程序便可从服务端socket对象上读取数据。

Android按键事件传递流程(二) - cheris_cheris的博客 - CSDN博客

1.当InputDispatcher通过服务端管道向socket文件描述符发送消息后，epoll机制监听到了I/O事件，epoll_wait就会执行返回发生事件的个数给eventCount，主线程开始执行epoll_wait后面的代码。

2.应用程序客户端通过NativeInputEventReceiver的InputConsumer方法从客户端管道InputChannel中获取事件消息，经过过滤，转化成应用层按键类型，再把按键事件传递到输入法窗口，应用层。

3.InputDispatcher先找到当前获得焦点的窗口，把事件发送给该窗口，窗口在启动activity时会创建，按键事件就传递到了获得焦点的窗口对应的所有view的根类DecorView，也可以说传递给了获得焦点的窗口对应的Activity对象。

4.如果有按键、触摸、轨迹球事件分发给Activity时，在具体事件处理之前，会回调onUserInteraction，一般情况下，用户需要自行实现该方法，与onUserLeaveHint一起配合使用辅助Activity管理状态栏通知。在按键按下、抬起时都会触发该方法回调；但在触摸时，只有触摸按下时被回调，触摸移动、抬起时不会回调。

5.应用层按键事件传递时涉及到很多情况，大概传递流程：

a. 应用层按键事件传递到view树根DecorView后分为两步：view树内部；view树外部(获得焦点的窗口)

如果view树内部没有消耗，就传递到view树外部，即传递给获得焦点的窗口的onKeyDown/onKeyUp；

b. view树内部一般先传递到当前Activity对象，如果没有消耗，传递到Activity的onKeyDown/onKeyUp；

c. Activity对象内部先分发给ViewGroup，viewGroup如果本身有焦点就传递给其父类view；

如果viewGroup本身没有焦点，就传递给其获得焦点的子view。子view分为两种情况：

如果子view是LinearLayout等常见布局，就递归传递过去，最后传递给获得焦点的view视图；

如果子view是纯粹的view视图，就传递给该视图；

d. view视图内部，如果设置了OnKeyListener监听器，就传递给OnKey；如果没有

OnKeyListener监听器，就分发给KeyEvent的dispatch，dispatch主要回调view的onKeyDown/onKeyUp；
e. 在view的onKeyDown/onKeyUp中，如果是DPAD_CENTER,KEYCODE_ENTER，直接处理；否则，更新绘制状态、执行长按处理、执行onClick方法等。
该小结没有考虑所有条件，只是大概给出传递流程，因为很多时候重写某个方法返回true不再传递下去，因此也就没有过多步骤。
6.特殊按键处理方法主要有：
interceptKeyBeforeQueueing
interceptKeyBeforeDispatching
PhoneWindow的onKeyDown/onKeyUp
PhoneFallbackEventHandler的dispatchKeyEvent
7.mFallbackEventHandler就是PhoneFallbackEventHandler对象，这是最后拦截特殊按键进行处理的方法，与 PhoneWindow区别的是，PhoneFallbackEventHandler针对所有窗口的。
8.按键事件的特殊处理可用逻辑执行顺序简单表达：
PhoneWindowManager的interceptKeyBeforeQueueing —->PhoneWindowManager的interceptKeyBeforeDispatching —-> PhoneWindow的onKeyDown/onKeyUp —-> PhoneFallbackEventHandler的dispatchKeyEvent
这个逻辑不是必须的，主要看按键是否消耗、按键的具体功能需求，有时候只需要在PhoneWindowManager中处理即可
9. 按键事件传递流程总结
a. InputReaderThread获取输入设备事件后过滤，保存到InboundQueue队列中
b. InputDispatcherThread从InboundQueue取出数据，先进行特殊处理，然后找到获得焦点的窗口，再把数据临时放到OutboundQueue中，然后取出数据打包后发送到服务端socket
c. 应用程序主线程中的WindowInputEventReceiver从客户端socket上读取按键数据，再传递给应用层
d. 应用层获取到事件后先分发给view树处理，再处理回退事件
10.输入事件核心组件
InputManagerService：输入事件的服务端核心组件，直接创建看门狗Watchdog、NativeInputManager对象，间接创建EventHub、InputManager以及

InputDispatcherThread、InputReaderThread线程，通过InputManager方法间接启动接收器线程、分发器线程；对系统特殊按键的处理；注册服务端管道
NativeInputManager：本地InputManager，创建c++层InputManager、EventHub对象
C++ 层InputManager：创建InputDispatcher、InputReader对象，创建并启动InputDispatcherThread、InputReaderThread线程
Java层InputManager：提供输入设备信息、按键布局等
InputReader：输入事件接收者，从EventHub中获得原始输入事件信息
InputDispatcher：分发事件给应用层，发送事件的服务端
EventHub：事件的中心枢纽，收集了所有输入设备的事件，包括虚拟仿真设备事件。
InputEventReceiver：接收输入事件的客户端
InboundQueue：InputReaderThread读取事件后保存到InboundQueue中等到InputReaderThread接收
outboundQueue：InputDispatcherThread从InboundQueue中取出数据放到outboundQueue中等待发送，然后从outboundQueue取出数据发送到服务端InputChannel等待应用层(或者称为客户端)接收
c++层InputChannel：一个输入通道，包含一对本地unix socket对象，服务端InputDispatcherThread向socket对象写入数据，客户端InputEventReceiver从客户端socket读取数据
Linux Input子系统分析之eventX设备创建和事件传递 - Wave的专栏 - CSDN博客
除了input keyevent 24，其他两种模拟发送键值的方法：
1.sendevent /dev/input/event0 1 114 1;sendevent /dev/input/event0 0 0 0;sendevent /dev/input/event0 1 114 0;sendevent /dev/input/event0 0 0 0;
2.通过Android jni代码向eventX写入事件，填充input_event结果然后写入/dev/input/event0。
android用户输入系统详细说明 - 七夜雪主 - CSDN博客
1.键扫描码Scancode是由Linux的Input驱动框架定义的整数类型。键扫描码Scancode经过一次转化后，形成按键的标签KeycodeLabel，是一个字符串的表示形式。按键的标签KeycodeLabel经过转换后，再次形成整数型的按键码keycode。在Android应用程序层，主要使用按键码keycode来区分。
2.kl按键布局文件，第1列为按键的扫描码，是一个整数值；第2列为按键的标签，是一个字符串。即完成了按键信息的第1次转化，将整型的扫描码，转换成字符串类型的按键标

签。第3列表示按键的Flag，带有WAKE字符，表示这个按键可以唤醒系统。
3.kcm按键字符映射文件，kcm表示按键字符的映射关系，主要功能是将整数类型按键码（keycode）转化成可以显示的字符。

Android 4.0 事件输入(Event Input)系统 - MyArrow的专栏 - 博客频道 - CSDN.NET
event对象关系图
InputMapper关系图：
触屏事件处理相关数据结构：
触屏事件处理流程：
InputReader::processEventsLocked设备增加、删除处理总结：
　　它负责处理Device 增加、删除事件。增加事件的流程为：为一个新增的Device创建一个InputDevice，并增加到InputReader::mDevices中；根据新增加设备的class类别，创建对应的消息转换器（InputMapper），然后此消息转换器加入InputDevice::mMappers中。消息转换器负责把读取的RawEvent转换成特定的事件，以供应用程序使用。
　　EventHub与InputReader各自管理功能：
　　1）EventHub管理一堆Device，每一个Device与Kernel中一个事件输入设备对应
　　2）InputReader管理一堆InputDevice，每一个InputDevice与EventHub中的Device对应
　　3）InputDevice管理一些与之相关的InputMapper，每一个InputMapper与一个特定的应用事件相对应，如：SingleTouchInputMapper。
[huasheng]apk_install 梳理相关文件

####性能优化
Android性能优化之Systrace工具介绍 | IT十万个为什么

####有关tombstone
Extracting variables from crash dump
Android NDK tombstone分析工具 - KoffuXu - 博客频道 - CSDN.NET
signal 11的定义在这里$android_root/prebuilts/gcc/linux-x86/host/x86_64-linux-glibc2.7-

4.6/sysroot/usr/include/bits/signum.h，通常crash也都是因为收到这个信号，也有少数是因为SIGFPE，即除0操作

stack.py工具

Android Native/Tombstone Crash Log 详细分析[原创]_夜莺_新浪博客

backtrace日志/addr2line/ndk-stack(相当于多次addr2line)/objdump(导出汇编代码)

Android 信号处理面面观 之 trace 文件含义 - 琪妙的Android世界 - CSDN博客

Trace文件是 android davik 虚拟机在收到异常终止信号 （SIGQUIT）时产生的。 最经常的触发条件是 android应用中产生了 FC （force close）。由于是该文件的产生是在 DVM 里，所以只有运行 dvm实例的进程（如普通的java应用，java服务等）才会产生该文件，android 本地应用 （native app，指 运行在 android lib层，用c/c++编写的linux应用、库、服务等）在收到 SIGQUIT时是不会产生 trace文件的。

1. 第一行是 固定的头, 指明下面的都是 当前运行的 dvm thread： “DALVIK THREADS:”
2. 第二行输出的是该 进程里各种线程互斥量的值。（具体的互斥量的作用在 dalvik 线程一章 单独陈述）
3. 第三行输出分别是 线程的名字（“main”），线程优先级（“prio=5”），线程id （“tid=1”） 以及线程的 类型（“NATIVE”）
4. 第四行分别是线程所述的线程组 （“main”），线程被正常挂起的次数（“sCount=1”），线程因调试而挂起次数（”dsCount=0“），当前线程所关联的java线程对象（”obj=0x400246a0“）以及该线程本身的地址（“self=0x12770”）。
5. 第五行 显示 线程调度信息。 分别是该线程在linux系统下得本地线程id （“sysTid=503”），线程的调度有优先级（“nice=0”），调度策略（sched=0/0），优先组属（“cgrp=default”）以及 处理函数地址（“handle=-1342909272”）

6 第六行 显示更多该线程当前上下文，分别是 调度状态（从 /proc/[pid]/task/[tid]/schedstat读出）（“schedstat=(15165039025 12197235258 23068)”），以及该线程运行信息 ，它们是 线程用户态下使用的时间值(单位是jiffies)（“utm=182”）， 内核态下得调度时间值（“stm=1334”），以及最后运行该线程的cup标识（“core=0”）；

7.后面几行输出 该线程 调用栈。

01_系统 常见 异常 分析 - ABPS(tombstone/anr/crash集合)

1.在mstar848上报的一个问题JASON-876，解析log

andbase@vm-10-58-57-52:/letv/workspace/mstar.848$ aarch64-linux-android-addr2line

-aCfe
out/target/product/sugarcane/obj/SHARED_LIBRARIES/libart_intermediates/LINKED/lib
art.so 00000000001ddb1c
0x00000000001ddb1c
art::gc::collector::ConcurrentCopying::MarkNonMoving(art::mirror::Object*,
art::mirror::Object*, art::MemberOffset)
art/runtime/gc/collector/concurrent_copying.cc:2435 (discriminator 2)
此处注释说，ref是个地址没有对齐的对象，此时堆一定被破坏了。
2432 if (is_los && !IsAligned<kPageSize>(ref)) {
2433 // Ref is a large object that is not aligned, it must be heap corruption. Dump
data before
2434 // AtomicSetReadBarrierState since it will fault if the address is not valid.
2435 heap_->GetVerification()->LogHeapCorruption(holder, offset, ref, /* fatal */
true);
2436 }
02_tombstone分析详解.pdf
03_ANR及watchdog分析详解.pdf
常见 core dump 原因分析signal 11 - SIGSEGV - CSDN博客
SIGABRT: 1.free多次 2.fclose多次
SIGSEGV: 3.引用空指针 4.fclose空指针
02_Core Dump（栈溢出，系统复位，SIGSEGV）_DEMETER-36086_TV卡在文件管理桌
面,2分钟后跳转开机动画【出现一次】 - ABPS【huasheng mark】
1.Process 2232 exited due to signal (11)表明system server是因为SIGSEGV导致的
crash。
2.接下来需要分析sp问什么被调整到一个不可访问的地址：
可能的原因有：a）在压栈和出栈的过程中，原有的sp记录被写错;
b）栈溢出
3.SignalHandler 在sigaction时已经添加了flag SA_ONSTACK，说明会在堆上分配
sigaltstack()。
4.通过kill -3给进程发信号打印trace，在系统忙的情况下可以触发signal 33
5.通过backtrace看了一下调用流程：

__android_log_vprint()中申请了一个1k的buffer -- char buf[LOG_BUF_SIZE];
接下来vsnprintf()定义了一些局部变量;
__vfprintf中定义了非常多的局部变量和结构数组
从以上调用流程看，android的log系统提供的函数不能在signal handler中调用。
我们在将stack调整到16k后，之前的复现方法不会再导致SIGSEGV，程序可以正常处理信号并打印log，说明的确是信号处理的栈空间不够支持打log。
6.解决此问题的方法一是扩大signal handler的stack，另外是杜绝在signal handler中打log。
由于改信号栈大小影响整个系统，最终采取的解决方案是去掉SignalHandler中的log输出。

####有关ANR
一、类型：
1. KeyDispatchTimeout（按键/触摸事件5s内没处理完）
2. BroadcastTimeout（onReceive 10s内没处理完）
3. ServiceTimeout（service的各个生命周期20s内没处理完）
二、原因：
1. 应用进程自身引起的，比如：主线程阻塞、挂起、死循环，执行耗时操作等；
2. 其他进程引起的，比如：其他进程CPU占用率过高，导致当前应用进程无法抢占到CPU时间片。常见的问题如文件读写频繁，io进程CPU占用率过高，导致当前应用出现ANR；
三、如何避免：
1. UI线程尽量只做跟UI相关的工作
2. 耗时的工作（比如数据库操作，I/O，网络访问，位图转换或者别的有可能阻碍UI线程的操作）把它放入单独的线程处理
3. 尽量用Handler来处理UIthread和别的thread之间的交互
4. 子线程尽量使用Android提供的API，比如HandlerThread，AsyncTask，AsyncQueryHandler等，这些API都提供了对于线程的系统级管理。如果应用直接使用Thread实现的话，则需要对这些子线程进行显式管理，比如线程池及线程周期的控制，以防止系统资源和内存泄漏；
5. BroadcastReceiver的onReceive（）方法执行完成后，BroadcastReceiver的实例就会被销毁。如果onReceive（）方法在10s内没有执行完毕，Android会认为改程序无响应。

所以在BroadcastReceiver里不能做一些比较耗时的操作，否则会弹出“Application NoResponse”对话框。特别说明的是，这里**不能使用子线程来解决，因为BroadcastReceiver的生命周期很短，子线程可能还没有结束BroadcastReceiver就先结束了**。BroadcastReceiver一旦结束，此时它所在的进程很容易在系统需要内存时被优先杀死，因为它属于空进程。**解决这个问题的方案是在onReceive()里开始一个Service，让这个Service去 做这件事情，那么系统就会认为这个进程里还有活动正在进行。**

四、所谓UI线程：

1. Activity:onCreate(), onResume(), onDestroy(), onKeyDown(), onClick(),etc
2. AsyncTask: onPreExecute(), onProgressUpdate(), onPostExecute(), onCancel,etc
3. Mainthread handler: handleMessage(), post*(runnable r), etc
4. other

五、分析log和trace，对案例分类

1. trace中main线程的调用栈比较干净，像下面这样，应该是cpu占用太高

【huasheng】看来这种trace对应cpu占用率太高（包括iowait太高）的情况

2. 访问网络时间太长
3. 申请图形相关内存失败的
4. 锁被其他线程长时间持有，或者干脆死锁状态的。

六、更多内容：

ANR原因分析及解决方法 - CSDN博客【huasheng mark】

ANR的产生需要同时满足三个条件：

* 主线程：只有应用程序进程的主线程响应超时才会产生ANR；
* 超时时间：产生ANR的上下文不同，超时时间也不同，但只要超过这个时间上限没有响应就会产生ANR；
* 输入事件/特定操作：输入事件是指按键、触屏等设备输入事件，特定操作是指BroadcastReceiver和Service的生命周期中的各个函数调用。

产生ANR的上下文不同，导致ANR原因也不同，主要有以下三种情况：

* 应用进程的主线程对输入事件在5s内没有处理完毕；
* 应用进程的主线程在执行BroadcastRecevier的onReceive函数时10s内没有处理完毕；
* 应用进程的主线程在执行Service的各个生命周期函数时20s内没有处理完毕；

为ANR主要是因为主线程由于耗时操作被阻塞而产生的，所以常见的解决方法是不在主线程做耗时操作，具体实现时需要注意以下几点：

· 主线程需要做耗时操作时，比如网络访问、数据库操作及位图变换等，必须启动一子线程处理，并利用handler来更新UI；
· 子线程尽量使用Android提供的API，比如HandlerThread，AsyncTask，AsyncQueryHandler等，这些API都提供了对于线程的系统级管理。如果应用直接使用Thread实现的话，则需要对这些子线程进行显式管理，比如线程池及线程周期的控制，以防止系统资源和内存泄漏；
· Broadcast Receiver中如果有耗时操作，可以放到service中来处理；
· 在后台子线程处理耗时操作时，为了提高用户体验，可以在前台界面显示某些动画或者progress bar；
如何分析解决Android ANR - CSDN博客【huasheng mark】
UI线程主要包括如下：
1. Activity:onCreate(), onResume(), onDestroy(), onKeyDown(), onClick(),etc
2. AsyncTask: onPreExecute(), onProgressUpdate(), onPostExecute(), onCancel,etc
3. Mainthread handler: handleMessage(), post*(runnable r), etc
4. other
ThreadState (defined at “dalvik/vm/thread.h “)
2. THREAD_UNDEFINED = -1, /* makes enum compatible with int32_t */
3. THREAD_ZOMBIE = 0, /* TERMINATED */
4. THREAD_RUNNING = 1, /* RUNNABLE or running now */
5. THREAD_TIMED_WAIT = 2, /* TIMED_WAITING in Object.wait() */
6. THREAD_MONITOR = 3, /* BLOCKED on a monitor */
7. THREAD_WAIT = 4, /* WAITING in Object.wait() */
8. THREAD_INITIALIZING= 5, /* allocated, not yet running */
9. THREAD_STARTING = 6, /* started, not yet on thread list */
10. THREAD_NATIVE = 7, /* off in a JNI native method */
11. THREAD_VMWAIT = 8, /* waiting on a VM resource */
12. THREAD_SUSPENDED = 9, /* suspended, usually by GC or debugger */
Android ANR分析 - 大雀儿飞飞的专栏 - CSDN博客(有关trace一些字段的说明)
anr问题分析 - ABPS
ANR 问题浅析[带pdf]
ANR问题分析指北 - 知乎专栏

1.常见的场景
A.input事件超过5S没有处理完成
B.service executing 超时（bind，create，start，unbind等等），前台20s，后台200s
C.广播处理超时，前台10S，后台60s
D.ContentProvider执行超时，20s
2.套路总结：
A. 查看traces.txt,如果有block栈则通过栈来找到block原因。
B. 应用自身耗时操作尝试修改异步，被Binder block则进一步确认下对端的情况，如果是依赖于外部状态的可能block操作，应用修改为异步，如果是common接口，转FW
C. 如果不存在栈，那么尽人事听天命，查看ANR点附近是否有主线程block的可能信息，如果有则顺腾摸瓜，没有就Needinfo。
3.如何避免ANR
A.减少复杂的layout
B.主线程尽量不要做和显示无关的事情。
C.如果存在可能会block、耗时的操作，不要放到主线程中，可以使用异步等方式来放到另外一个handlerthread或者asynctask中去做。
需要注意的是，使用handler.post或者sendmessage的时候需要确认清楚handler的looper，如果是主线程，依然可能会ANR。
在HandlerThread中如果需要处理和显示相关的，还需要到主线程中处理（非UI线程不能操作UI，在ViewRoot中有检查）
03_ANR及watchdog分析详解.pdf

####有关watchdog的分析
watchdog分两类，monitor checker监测对象是否发生死锁和looper checker监测注册的消息队列是否长时间被占用。
其注册方法分别为addMonitor、addThread，AMS举例如下：
发生watchdog时候，类型二，monitor checker超时，从trace中查找android.fg所在的地方；类型二，looper checker超时，从trace中查找looper所在的地方，“Blocked in handler on ActivityManager”

更多详情见：Watchdog机制以及问题分析 - CSDN博客【huasheng mark】

03_watchdog_Blocked in handler_DEMETER-38318_In Le advertising video interface to enter the STR, wake-up after the TV restart (在Le播放视频的广告界面进入STR,唤醒后电视重启)

04_watchdog_Blocked in monitor_DEMETER-32752_TV automatically restart the boot animation into the dead cycle（电视自动重启开机动画进入死循环）

tombstone里的错误并不会引起重启，属于常见的问题。引起system_server watch dog的是因为并发的binder数量一直超过或等于最大的数量

有很多这样的log:Waiting for thread to be free. mExecutingThreadsCount=16 mMaxThreads=15 卡在了 IPCThreadState::blockUntilThreadAvailable().

05_Memory corruption issue_IRIS-13030 开机广告卡顿，system_server重启问题 - ABPS—Note.txt

gdb需要好好玩一玩

05_Memory corruption issue_IRIS-13030 开机广告卡顿，system_server重启问题 - ABPS【huasheng mark??】

06_Software watchdog blocked when dumpstate-DEMETER-47450-进行MTBF测试中运行不同case时频繁发生system server crash - ABPS【huasheng mark】

07_surfaceflinger Crash（系统复位，SIGSEGV）_DEMETER-43560_从信号源桌面切换到live桌面，分享键截屏，电视重启

从调用路径上看是captureScreenWithClip，从函数名上看，截屏是带有裁剪的，但之前看到截屏图像使用的copy函数是对连续内存操作的，如果发生裁剪，那么截取的内存肯定不是连续的（图像在内存中数逐行连续存放的，如果截取中心部分，那内存截取就不是整行，所以不连续）

08_Memory leak in Native layer_DEMETER-48786 进行MTBF测试时，电视发生死机，无法正常运行

09_MemoryHeapBase and fd Leak_DEMETER-52234 在进行MTBF测试时，电视卡在购物桌面，发生死机无法进行操作

Watchdog机制以及问题分析 - CSDN博客【huasheng mark】

1.Android的Watchdog是一个单例线程，在System Server时就会初始化Watchdog。

Watchdog在初始化时，会构建很多HandlerChecker，大致可以分为两类：

* Monitor Checker，用于检查是Monitor对象可能发生的死锁, AMS, PKMS, WMS等核心

的系统服务都是Monitor对象。
* Looper Checker，用于检查线程的消息队列是否长时间处于工作状态。Watchdog自身的消息队列，Ui, Io, Display这些全局的消息队列都是被检查的对象。此外，一些重要的线程的消息队列，也会加入到Looper Checker中，譬如AMS, PKMS，这些是在对应的对象初始化时加入的。
2.两类HandlerChecker的侧重点不同，Monitor Checker预警我们不能长时间持有核心系统服务的对象锁，否则会阻塞很多函数的运行; Looper Checker预警我们不能长时间的霸占消息队列，否则其他消息将得不到处理。这两类都会导致系统卡住(System Not Responding)。
3.WindowManagerService实现了Watchdog.Monitor这个接口，并将自己作为Monitor Checker的对象加入到了Watchdog的监测集中；monitor()方法是运行在android.fg线程中的。Android将android.fg设计为一个全局共享的线程，意味着它的消息队列可以被其他线程共享， Watchdog的Monitor Checker就是使用的android.fg线程的消息队列。因此，出现Monitor Checker的超时，肯定是android.fg线程阻塞在monitor()方法上。
4.关于Binder线程。当Android进程启动时，就会创建一个线程池，专门处理Binder事务。线程池中会根据当前的binder线程计数器的值来构造新创建的binder线程, 线程名”Binder_%X”，X是十六进制。当然，线程池的线程数也有上限，默认情况下为16，所以，可以看到 Binder_1 ~ Binder_F 这样的线程命名。
5.在多核情况下，CPU的使用率统计会累加多个核的使用率，所以会出现超过100%的情况。
6.对于压力测试而言，我们一般会设定一个通过标准，在某些压力情况下，出现一些错误是允许的。对于Android实际用户的使用场景而言，本例中的压力通常是不存在的，所以在实际项目中，这种类型的Watchdog问题，我们一般不解决。
7.Android中Watchdog用来看护system_server进程，system_server进程运行着系统最终要的服务，譬如AMS、PKMS、WMS等， 当这些服务不能正常运转时，Watchdog可能会杀掉system_server，让系统重启。

####一个watchdog实例
[IRIS-6461]【Max55 Blade_dailybuild_0624 live桌面】【dailybuild】【ET】在轮播6频道按遥控器51频道，界面卡在6频道，遥控器无响应。（出现一次）

最终在WindowStateAnimator中的relayoutWindow中，锁住mWindowMap，迟迟没有释放。

最终有mstar从google源码中找到个patch解决此问题：

####

PREOPT、ABI配置的研究，及现有TVUI应用问题改进 -Wiki

1.新方案思路为：不破坏源生逻辑，对共用了system uid、account uid的应用建立白名单机制，使白名单里的应用如果没有jni库则运行在OS默认的指令集，即64位机上直接由zygote64启动，就不会出现问题提出描述的Dex找不到问题。

2.PREOPT参数：

```
# true的时候开启DEX全局预编译优化
WITH_DEXPREOPT := true
# true默认选项，开启预编译，且删除class.dex
# nostripping开启预编译，不删除class.dex
# false不开启预编译
DEX_PREOPT_DEFAULT ?= true
# false表示关闭DEX优化
LOCAL_DEX_PREOPT := false
```

3.出现Dex找不到问题的根源是因为：不同应用过多使用相同的sharedUserId，特别是android.uid.system。所以针对EUI、BSP各系统应用这里有几点建议：

1）系统应用如果不用system权限的，可以不用system uid，不用相同uid的应用可以尽量不用。

2）各应用尽量不要主动关闭应用DEX优化（LOCAL_DEX_PREOPT := false），会增加系统启动时间，影响应用启动性能，也会增加/data分区占用。

3）应用自带的Jni库建议优先支持64位，最好能同时兼容32、64位库。

####

研发流程及分工 - TV-EUI研发Wiki

####资源及代码overlay
overlay机制 - Wiki

####如果父进程退出，本进程也需要退出
子进程中设置prctl(PR_SET_PDEATHSIG, SIGKILL); 表示父进程退出则会收到SIGKILL信号。

####执行hprof时可能出现按键无响应
通过ddms执行hprof，android 会暂停，此时出现ANR.WATCHDOG都有可能。此时出现的卡顿，延迟，ANR/watchdog 很正常。
往往还伴随着adbd占用cpu率很高。

####支持保存多个traces.txt文件

####traces.txt中添加binder信息
得到的信息举例如下：

####有关binder
1.有关Ixxx.Stub()

####有关UID

####

Android中Context详解 ---- 你所不知道的Context - qinjuning、lets go - 博客频道 - CSDN.NET

1.Context概念可知:

1.1 它描述的是一个应用程序环境的信息，即上下文。

1.2 该类是一个抽象(abstract class)类，Android提供了该抽象类的具体实现类(后面我们会讲到是ContextIml类)。

1.3 通过它我们可以获取应用程序的资源和类，也包括一些应用级别操作，例如：启动一个Activity，发送广播，接受Intent信息 等。。

2.相关类继承关系：

3.应用程序在什么情况需要创建Context对象的？应用程序创建Context实例的情况有如下几种情况：

1、创建Application 对象时， 而且整个App共一个Application对象

2、创建Service对象时

3、创建Activity对象时

因此应用程序App共有的Context数目公式为：

总Context实例个数 = Service个数 + Activity个数 + 1（Application对应的Context实例）

####有关运行时权限申请

1.在Android 6.0（API level 23）之后，用户是在应用运行时才去授予权限给应用，而不是在用户安装app的时候。这种方法简化了应用的安装过程，因为用户无需在应用安装或升级时去授予权限给应用。这也可以让用户更有效地监控应用的功能。例如，用户可以选择给相机应用一个访问相机的权限，而不给相机应用指定定位的权限。去到应用的设置界面，用户是可以随时撤销这个权限的。

系统权限被分为两类：normal 和 dangerous：

1>normal 权限不会直接涉及用户的隐私。**如果你的应用在manifest中列出这个normal权**

限，那么系统会自动同意这个权限。
2>dangerous权限能够让应用去访问用户的隐私数据。如果你的应用在manifest中列出了一个normal权限，系统会自动同意这个权限，**如果你在manifest中列出了一个dangerous权限，则需要征求用户的批准。**

####有关弹框申请权限
问题：互娱那边一个游戏apk，在938上运行直接crash，在648上会弹框申请权限，在允许后能正常运行。
调试：1. 运行requestPermissions READ_CONTACTS在两个平台上都会crash，log中显示数组空指针
01-25 19:55:38.812 E/AndroidRuntime(15923): Caused by:
java.lang.NullPointerException: Attempt to get length of null array
01-25 19:55:38.812 E/AndroidRuntime(15923): at
com.android.packageinstaller.permission.ui.GrantPermissionsActivity.computePermissionGrantState(GrantPermissionsActivity.java:277)
2.调用shouldShowRequestPermissionRationale，在938/648上都返回false
3.调用TelephonyManagerget的DeviceId，在AndroidManifest.xml中声明了权限，在938上crash，在648上不报错不弹框返回值为null。
01-26 10:21:58.194 E/AndroidRuntime(21612): java.lang.RuntimeException: Unable to start activity ComponentInfo{com.example.myhw5/com.example.myhw5.MainActivity}: java.lang.SecurityException: getDeviceId: Neither user 10067 nor current process has android.permission.READ_PHONE_STATE.
4.使用Manifest.permission.READ_CONTACTS在938/648上一直不能弹框，换了Manifest.permission.READ_PHONE_STATE在648上requestPermissions能弹框了。
5.确实需要requestPermissions才能弹框，在648上不调用也不会弹框，在AndroidManifest中申请了就能调用成功。在938上不调用getDeviceId也能弹框，看来问题不在能不能弹框上，而在调用getDeviceId上，调用就crash，来不及显示出来弹框。
6.将调用getDeviceId的操作放在onRequestPermissionsResult中，938上能弹框并且不会crash，看来requestPermissions弹框操作是在新线程中做的，不等弹框并用户选择后代码继续往下执行。

####settingadaptor
1.AudioMode.java作为一个存储数据的类不用AudioMode.aidl来生成了，但AudioMode.aidl还是需要，要不然编译失败。

2.有关提供的app开发的sdk jar包，默认编译不生成classes.jar，是odex优化后的classes.dex。
在Android.mk中添加LOCAL_JACK_ENABLED := disabled 可使得编译生成classes.jar，解压后其中有相应的class。
3.一切编译ok后，将jar替换到/system/framework/重启电视，SettingAdapterService.apk、GlobalSetting.apk这两个apk直接覆盖式安装即可。
adb push out/target/product/mangosteen/system/priv-app/SettingAdapterService/SettingAdapterService.apk /system/framework/
adb install -r out/target/product/mangosteen/system/priv-app/SettingAdapterService/SettingAdapterService.apk
adb install -r out/target/product/mangosteen/system/app/GlobalSetting/GlobalSetting.apk

dumpsys activity
查看taskrecord

####pstree 查看应用内部线程

####有关匿名共享内存
Android系统匿名共享内存子系统Ashmem两个特点，一是能够辅助内存管理系统来有效地管理不再使用的内存块，二是它通过Binder进程间通信机制来实现进程间的内存共享。第

二个特点我们在上面这个例子中看到了，但是似乎还不够深入，我们知道，在Linux系统中，文件描述符其实就是一个整数，它是用来索引进程保存在内核空间的打开文件数据结构的，而且，这个文件描述符只是在进程内有效，也就是说，在不同的进程中，相同的文件描述符的值，代表的可能是不同的打开文件，既然是这样，把Server进程中的文件描述符传给Client进程，似乎就没有用了，但是不用担心，在传输过程中，Binder驱动程序会帮我们处理好一切，保证Client进程拿到的文件描述符是在本进程中有效的，并且它指向就是Server进程创建的匿名共享内存文件。至于第一个特点，我们也准备在后续学习Android系统匿名共享内存子系统Ashmem时，再详细介绍。

更多详情见： Android系统匿名共享内存Ashmem（Anonymous Shared Memory）简要介绍和学习计划 - 老罗的Android之旅 - CSDN博客